# Lab 9 - Dynamic Memory & Classes

## Goals

By the end of this lab you should:

- Know how to write a copy constructor, an overloaded assignment operator, and an overloaded stream insertion operator.
- Understand the difference between shallow vs deep copies.

---

## Introduction

All this time working with classes that have member variables that are pointers, we've been skating on thin ice. We've been risking memory leaks and seg faults because we did not fully show you how to manage those pointers through the life of an object instantiated from the class.

Yes, we showed you how to write destructors - but it turns out that that's usually not enough!
We need ... ***"the Big Three!!"***

---

## Destructors

We have seen that a special member function, the constructor, is called automatically when an object is first created. Likewise, when an object goes out of scope or is destroyed, another special member function called the destructor is called automatically. A destructor has the same name as the constructor (which is the same as the class name ), but it is preceeded by a tilde (~). Here is an example:

```
class Test
{
 public:

   // Constructor (same name as the name of the class)
   Test()
   {
      data = new int(10);
   }

   // Destructor (same name , but preceeded by a tilde (~))
   ~Test()
   {
      delete data;
   }
```

```
 private:
   int *data;
};
```

The job of a destructor is to free up the memory that is taken up by an object when we no longer need it. If you don't provide a destructor in your class definition, the compiler will automatically create a simple default destructor for you, which is perfectly fine if none of the member variables are pointers - but quite useless otherwise!

---

## Copy Constructors

We know that we can define and initialize an object to the value of another object in a single statement. Let's say we have a class Area. We can say:

```
// Copy initialization - we're assuming that Area object a2 already
// has a well-defined value
Area a3(a2); //a3 is now an exact copy of the object a2
```

or we can also say:

```
// Alternate syntax, identical action
Area a3 = a2;
```

Both styles of declaration/initialization invoke the ***copy constructor*** - i.e. a constructor that copies its arguments into a ***new*** object. The default copy constructor, which is provided automatically by the compiler for every object, performs a member-by-member copy (aka ***shallow copy***). This is similar to what the assignment operator does (see below): the difference is that the copy constructor ***also creates a new object***.

This is not the only time a copy constructor is invoked: think about what has to happen when an argument is passed into a function ***by value*** or an object is returned by a function ***by value***. The copy constructor is invoked here too.

As with the destructor and the default constructor, if you don't explicitly define your own copy constructor, the compiler will define it for you.
And as with the destructor, if your class involves pointers, the default copy constructor will usually be useless (or even dangerous!) In this case, you need to define your own copy constructor that copies the objects in the heap (aka **deep copy**).

---

## The Assignment Operator

You've used the assignment operator (=) many times, probably without thinking too much about it. We've used it to assign a new value to variables:

```
int a;
... // a is given a value & used for some stuff ..
a = 67;
```

We've used it to assign the value of one variable to another existing variable:

```
int b;
... // b is given a value & used for stuff
b = a;
```

Note that this use of the assignment operator is ***quite different*** from that of the examples in the previous section, where a variable was being ***constructed and initialized*** in the same statement.

Here, the left-hand object already exists as large as life, and is just getting a new value: that's the job of the assignment operator.

```
Area a3; // Area object constructed from the default
         // constructor
Area a2(args); // Area object constructed from constructor
               // with parameters
a3 = a2; // the area object a3 gets re-assigned to the value
         // of a2
```

As before, if you don't explicitly define your own overloaded assignment operator, the compiler will do it for you - and as before, if your class involves pointers, it will probably do it wrong!!

Consider the following small (and incomplete) example using a string class that wraps a c-style string in order to give functionality similar to the string class from the standard template library.

```
class StringVar
{
 public:
  .
  .
  .

   // Overloads the assignment operator =
   // to copy a string from one object to another.
   StringVar & operator=( const StringVar &right_side );

  .
  .
  .
};

StringVar & StringVar::operator=( const StringVar &right_side )
```

```
{
   if (this != &right_side)
   {
      int new_length = strlen(right_side.value);
      if (new_length > max_length)
      {
         delete [] value;
         max_length = new_length;
         value = new char[max_length + 1];
      }
      for (int i = 0; i < new_length; i++)
      {
         value[i] = right_side.value[i];
      }

      value[new_length] = '\0';
   }
   return *this;
}
```

The assignment operator can now be used just as you always use the assignment operator:
string1 = string2;

*Notice that the argument to operator =() is passed by reference. It is not strictly necessary to do this, but is nearly always a good idea.*
*An argument passed by value generates a copy of itself in the function to which it is passed: such objects can sometimes be quite large, wasting a lot of time and memory. So as we have always taught you, all object parameters should be passed by reference (or const reference), unless there is an exceptionally good reason for passing by value.*

The copy constructor, the assignment operator ("=") and the destructor are called ***the big three*** **because if you need to define any one of them, then you need to define all three.**

**For any class that uses pointers initialized by the new operator, you should always define your own copy constructor, overloaded =, and destructor.**

---

---

## Programming Exercise

Implement the following class design:

```
class Music_collection
{
 private:
   int number; // the number of tunes actually in the collection
   int max;    // the number of tunes the collection will ever be able to hold
```

```
   Tune *collection; // a dynamic array of Tunes: "Music_collection has-many Tunes"

 public:
   // default value of max is a conservative 100
   Music_collection();

   // sets max to n
   Music_collection( int n );

   // overloaded copy constructor
   Music_collection( const Music_collection &m);

   // returns true if add was successful,
   // returns false if not enough room to add
   bool add_tune( const Tune &t );

   // sets the Tune at position index in collection to t,
   // returns true if index < number
   bool set_tune( int index, const Tune &t );

   // overloaded assignment operator
   Music_collection & operator=( const Music_collection &m );

   // Destructor
   ~Music_collection();

  // overloaded stream insertion operator
  // outputs the title of each Tune in the collection on a separate line
  friend ostream & operator<<( ostream &out, const Music_collection &m );
};
```

The Tune class is ultra-simple:

```
class Tune
{
  private:
   string title;

  public:
   Tune();
   Tune( const string &n );
   const string & get_title() const;
};
```

---

# Testing

Now let's give your code a test run.

You should build a test harness that uses the following pseudo-code:

```
main
{
  create a few Tune objects to use later on;
  Music_collection A;
  add a few tunes to A;
  Music_collection B(A);
  change a Tune in B using set_tune function;
  Music_collection C;
  C = B;
  add a Tune to B;
  change a Tune in C using set_tune function;
  print A,B,C;
}
```

After you test your work, comment out the assignment operator and copy constructor in .h and .cpp files of your `Music_collection` class implementation. re-compile and run your code. Compare the results with the previous run. (Ignore seg. fault and memory corruption errors, on terminal just check the lines right after you call your binary executable, i.e a.out)
Pay careful attention to the "boundary" or special cases: e.g. what happens if you try to assign an object to itself? *(When could this happen?)*

Remember to free the memory of the left-hand-side object of the assignment operator.
Remember to make ***deep*** copies, not ***shallow*** copies. *(Make sure you can explain exactly what this means!)*

---

# What to submit

Submit the following files:

- Music_collection.h (this file also contains Tune class)
- Music_collection.cpp
- main.cpp